Title	Wait-Free Linearizable Implementation of a Distributed Shared Memory (Algorithm Engineering as a New Paradigm)
Author(s)	Inoue, Michiko; Suda, Katsuro; Moriya, Sen; Masuzawa, Toshimitsu; Fujiwara, Hideo
Citation	数理解析研究所講究録 (1999), 1120: 78-87
Issue Date	1999-12
URL	http://hdl.handle.net/2433/63492
Right	
Type	Departmental Bulletin Paper
Textversion	publisher

数理解析研究所講究録
1120巻 1999年 78-87

線形化可能な分散共有メモリの無待機な実現

Wait-Free Linearizable Implementation of a Distributed Shared Memory

井上 美智子　須田 克朗[1]　守屋 宣
Michiko INOUE　Katsuro SUDA　Sen MORIYA
増澤 利光　藤原 秀雄
Toshimitsu MASUZAWA　Hideo FUJIWARA

奈良先端科学技術大学院大学
〒630-0101 奈良県生駒市高山町 8916-5
{kounoe, katuro-s, sen-m, masuzawa, fujiwara}@is.aist-nara.ac.jp

Abstract: We consider wait-free linearizable implementations of shared objects which tolerate crash faults of any number of processes on a distributed message-passing system. We consider the system where each process has a local clock that runs at the same speed as real-time clock and all message delays are in the range $[d-u, d]$ where d and u $(0 < u \leq d)$ are constants known to every process. We present four wait-free linearizable implementations of read/write registers and two wait-free linearizable implementations of general objects for several system settings. These are the first implementations with taking account of wait-freedom. Moreover, the worst-case response times of our wait-free implementations of read/write registers on a reliable broadcast model is better than any previously known implementations.

Keywords: message-passing system, distributed shared memory, linearizability, wait-freedom

1 Introduction

How to provide logically shared objects in a distributed system is a fundamental problem on concurrent computing. A distributed system with shared objects has good scalability while it needs low-level or complex control to shared data through message-passing paradigm. Logically shared objects greatly simplifies a design of a user program thanks to its simple and general computing paradigm. A *distributed shared memory* consisting of such shared objects aims at providing useful and scalable programming environment for high-performance computing using multiple processors.

We implement logical shared objects which are used by multiple application processes concurrently. The implemented shared objects should provide some consistency for concurrent accesses. We consider *linearizable* implementations [1] of shared objects on a distributed message passing system. Informally, linearizability guarantees that operations to the implemented objects seem to be executed sequentially in some total order, and, for two operations such that one operation starts after the other operation completed, this total order preserves the real-time order on them. It has some good properties, such as *locality* and *nonblocking*. Locality means that a system is linearizable if each individual object is linearizable. Locality allows concurrent system to be designed and constructed in a modular fashion; each of linearizable objects can be implemented, verified and executed independently. Nonblocking property means that a pending operation is never required to wait for another pending operation to complete. Nonblocking implies that linearizability is an appropriate condition for a system where real-time response is important.

An implementation is said to be *wait-free* if any operations of the implemented object are com-

[1]現在，NTTソフトウェア.

Table 1: Linearizable implementations (Processes do not crash).

read/write register

clock	$res\_time(write)$	$res\_time(read)$	
async.	$\geq u/2$		[2]
	$res\_time(write) + res\_time(read) \geq d + u/2$		[3]
		$\geq u/2$	[3]
u-sync.	$\leq (1-\beta)d + 3u$ $(0 \leq \beta < (d-u)/d)$	$\leq \beta d + 4u$	[3]

general object

async.	$res\_time(op_a)$	$res\_time(op_b)$	
async.	$\leq u$	$\leq 2d$	[4]

Table 2: Wait-free linearizable implementations in this paper (Processes may crash).

read/write registers, reliable broadcast

clock	$res\_time(write)$	$res\_time(read)$
async.	d	u
u-sync.	$u + \alpha \cdot A$ $(A = \max\{d-2u, 0\},\ 0 \leq \alpha \leq 1)$	$u + (1-\alpha)A$

read/write registers, unreliable broadcast

async.	d	d
u-sync.	u	d

general objects, reliable broadcasts

	$res\_time(op_a)$	$res\_time(op_v)$
async.	u	$2d$
u-sync.	u	$d+u$

pleted in finite time regardless of other processes' behavior[5]. We consider wait-free implementations, which tolerates crash faults of any number of processes. James et al. showed that there are no wait-free linearizable implementations of read/write registers on the system in which message delays are arbitrary[6]. In this paper, we assume that all message delays in the system are in the range $[d-u, d]$ for some constants d and $u (0 < u \leq d)$ where every process knows these d and u, and the system provides each process with a local clock that runs at the same rate as global time. We consider two kinds of models about exchange of messages, a *reliable broadcast* and an *unreliable broadcast*. These two models differ in a guarantee on a case where a process crashes during its broadcast. A broadcasted message is guaranteed to be received by all correct processes in a reliable broadcasted model. On the other hand, in an unreliable broadcast model, if a process crashes during its broadcast, the message is not guaranteed to be received by all processes. We consider two kinds of models also on local clocks, *asynchronous clocks* and *u-synchronous clocks*. In a u-synchronous clock model, the difference between any pair of two local clock values is at most u. In an asynchronous clock model, we make no assumptions on such a difference. The efficiency of an implementation is measured by the worst-case response time $res\_time(op)$ for each operation op of the implemented objects.

Several authors have investigated linearizable implementations of shared objects on a system in which no processes crash and all message delays are in the range $[d-u, d]$ (Tab. 1). In Tab. 1, op_a is any operation returning a unique response, called to be *ack-type*, and op_v is any operation that is not ack-type, called to be *val-type*. In this paper, we consider wait-freedom and present six wait-free linearizable implementations shown in Tab.2. The response time of our wait-free implementation of read/write registers on a reliable broadcast and u-synchronous model is better than the previously known implementation in [3].

2 Definitions

2.1 System

A distributed message-passing system consists of multiple processes and a communication network. A process communicates with any other processes by exchanging messages through the network. All message delays are in the range $[d-u, d]$ for some constants d and u $(0 < u \leq d)$ known to every process. Each process has a local clock that runs at the same rate as global time [2]. The process obtains local time from its local clock and

[2] We use system-wide global time to specify system behavior. Note that the global time is introduced only for specification and no processes can use it.

uses a timer based on its local clock. We assume the difference between any pair of local clock values in a system is at most ϵ for some constant ϵ. Such a model is called *ϵ-synchronous clock* model. If ϵ is the infinity, the model is called *asynchronous clock* model. We assume that a process may crash. After a process crashes, it ceases to operate. We consider two kinds of models about exchange of messages, a *reliable broadcast model* and an *unreliable broadcast model*. In a reliable broadcast model, if a process crashes during broadcasting a message, it is guaranteed that all correct processes receive it or no process receives it.

A process p is modeled as a state machine. Its state changes when some event occurs at p. A *system configuration* (or we call just *configuration*) is defined as all process states, a set $\mathcal{N}$ of in-transit messages and sets $\mathcal{A}_p$ of alarms which have been set to its timer and have not gone off at each process p. An in-transit message is a triple (M, s, r) where M is a message, s is the sender, and r is the destination. An alarm is a pair (K, t) where K is the type of an alarm and t is the local time for the alarm to go off. A process can set multiple alarms concurrently, and a type of alarm is used to identify them. Each process has the following events.

- Communication events: Broadcast events $BroadCast(p, M)$ and receive events $Receive(p, q, M)$ can occur in a reliable broadcast model. Send events $Send(p, q, M)$ and receive events $Receive(p, q, M)$ can occur in an unreliable broadcast model, .
  - $Send(p, q, M)$: Process p sends a message M to process q. A triple (M, p, q) is added to $\mathcal{N}$.
  - $BroadCast(p, M)$: Process p broadcasts a message M [3]. For any process q, (M, p, q) is added to $\mathcal{N}$.
  - $Receive(p, q, M)$:Process p receives a message M from process q. A triple (M, q, p) is removed from $\mathcal{N}$.
- Time events:
  - $TimerSet(p, \bar{t}, K)$: Process p sets its timer of type K to go off after $\bar{t}$. When an event $TimerSet(p, \bar{t}, K)$ occurs at local time t, a pair $(K, t + \bar{t})$ is added to $\mathcal{A}_p$.
  - $Alarm(p, K)$: An alarm of type K occurs at process p. When an event $Alarm(p, K)$ occurs at local time t, a pair (K, t) is removed from $\mathcal{A}_p$.
  - $ReadClock(p, s)$: Process p obtains the clock value s from its local clock.
- $Stop(p)$: Process p crashes. After this event, p's state changes to *fault state* and p ceases to operate.
- A process communicates also with the outside of the system, which we call *environment*. We describe events about communication between a process and environment later.

The receive, alarm and stop events are *input* events, which arise passively.

The *system history* (or we call just *history*) is defined as a finite or infinite alternating sequence of configurations and occurrences of events, $H = c_0, (e_1, T_1), c_1, \cdots, c_k, (e_{k+1}, T_{k+1}), c_{k+1}, \cdots$, where each $c_k (k \geq 0)$ is a configuration, $(e_k, T_k) (k \geq 1)$ is an occurrence of an event, e_k is the event and T_k is global time when e_k occurs. Each T_k is denoted by $time(e_k)$. A process p's state is a projection of a configuration c_k to p, denoted by $c_k|p$. The first configuration c_0 is called an initial configuration, in which all processes are in the initial state, and $\mathcal{N}$ and $\mathcal{A}_p$ for any process p are empty. For each k, $T_k \leq T_{k+1}$ holds. A history H implies that, for each $k (k \geq 0)$, an event e_{k+1} occurs at some process p at T_{k+1} in a configuration c_k, and p's state changes from $c_k|p$ to $c_{k+1}|p$ (and $\mathcal{N}$ or $\mathcal{A}_p$ also may change). To be simplified, all events in a history are distinct. We assume the following conditions on any history H.

- If $\mathcal{A}_p$ contains a pair (K, t), $Alarm(p, K)$ occurs or p is in a fault state at local time t. Conversely, $Alarm(p, K)$ occurs at t, only if (K, t) is in $\mathcal{A}_p$.
- If a triple (M, p, q) is added to $\mathcal{N}$ at global time T, $Receive(q, p, M)$ occurs in $[T + d - u, T + d]$ or p is a fault state at T. Only if

[3] For convenience, we assume that a process sends a message to all processes including itself by a broadcast.

$\mathcal{N}$ contains a triple (M,p,q), a receive event $Receive(q,p,M)$ occurs.

2.2 Implementation of an object

We define a *deterministic shared object* (we call just *object* in the following). An object is a data structure to which multiple processes can access concurrently. An object has a unique *name* and a *type*. The type is a tuple $(OP, RES, Q, q_0, \delta)$, where OP is a set of operations, RES is a set of responses, Q is a set of states, q_0 is an initial state, and $\delta : Q \times OP \rightarrow Q \times RES$ is a function called *sequential specification*. The sequential specification defines a behavior of the object when operations are applied sequentially: if an operation op is applied to the object in a state s, the object changes its state to s' and returns the response res where $\delta(s, op) = (s', res)$ holds. Such an object is called to be *deterministic*, since the sequential specification is a function. If an operation op always returns a unique response, that is $|\{res|\exists s, s'[\delta(s, op) = (s', res)]\}| = 1$, op is called to be *ack-type*. An operation is called to be *val-type*, if it is not ack-type. In the following, op_a denotes any ack-type operation and op_v denotes any val-type operation.

Next, we define an *implementation* of an object O of type $(OP, RES, Q, q_0, \delta)$. We implement a virtual shared object which is used concurrently by environment. Figure 1 illustrates the implementation. An object is implemented by a set of processes $\{p_1, p_2, \cdots, p_n\}$. A subscript i of each p_i is the process identifier. Environment can access an object by communicating with a process p_i. Communication between environment and p_i is modeled as the following events.

- $Invoke(p_i, op)$: Environment calls p_i to apply an operation $op(\in OP)$ to the object O.
- $Response(p_i, res)$: Process p_i returns a response $res(\in RES)$ for an invocation to environment.

The invoke event is an input event. We assume the following condition about communication between environment and p_i on any history H.

- Once environment invokes an operation to a process p_i, it does not invoke the next operation to p_i until p_i returns a response for the former invocation.

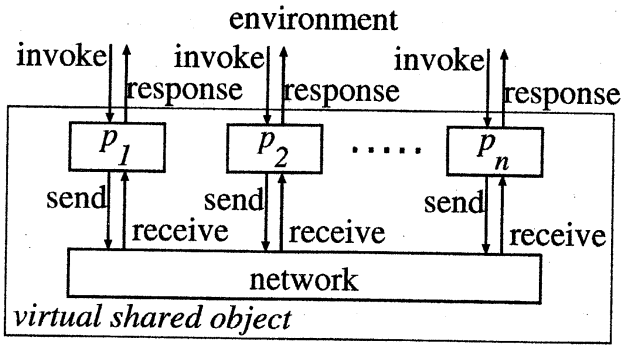

Figure 1: Implementation of a shared object.

To return consistent responses, the processes may exchange messages with each other.

For each p_i, we consider the restricted sequence of a history H to p_i's invoke and response events. It should be an alternating sequence $Inv_1, Res_1, Inv_2, Res_2, \cdots$ where Inv_k is an invoke event and Res_k is a response event for each $k(k \geq 1)$. For each invoke event Inv_k, the next event Res_k is called to be a *corresponding* response event. A pair of events (Inv_k, Res_k) is called an *operation execution*. An invoke event that has no corresponding response events is said to be *pending*. If an invoke event of an operation is not pending, the operation execution is said to be *completed*.

We adopt *linearizability* as a consistency condition of an implementation of an object. Herlihy et al. showed a *local* property of linearizability[1]. The locality means that an implementation of multiple shared objects is linearizable if and only if each object is implemented linearizably. In this paper, we consider an implementation of one object, and we define an implementation of only one object. From locality, we can implement multiple objects from our implementations of one object.

Now we define linearizability and wait-freedom. We consider a sequence of operation executions $\tau = (Inv_1, Res_1), (Inv_2, Res_2), \cdots$. For each $k(k \geq 1)$, let Inv_k and Res_k be $Invoke(p_{i_k}, op_k)$ and $Response(p_{i_k}, res_k)$ respectively. For an object O of type $(OP, RES, Q, q_0, \delta)$, if there exists a sequence $\theta = q_0, q_1, \cdots$ of states of O, where $\delta(q_{k-1}, op_k) = (q_k, res_k)$ holds for each $k \geq 1$, τ is said to be *legal*. In a system history H, if $time(Res_k) < time(Inv_l)$ holds for

two operation executions $op_k = (Inv_k, Res_k)$, $op_l = (Inv_l, Res_l)$, we say that op_k precedes op_l, denoted by $op_k \xrightarrow{H} op_l$. A restricted sequence of H to completed invoke events and response events is denoted by $complete(H)$.

Definition 1 *A history H is said to be linearizable, if there exists a history H' that satisfies the followings.*

- *The history H' is obtained from H by appending corresponding response events for some (possibly empty) pending invoke events.*
- *There exists a legal sequence τ consisting of all operation executions in $complete(H')$ such that, for any operation executions op_1 and op_2 satisfying $op_1 \xrightarrow{complete(H')} op_2$, op_1 precedes op_2 in τ.* ■

Definition 2 *An implementation I is said to be linearizable if any possible system history H is linearizable.* ■

Definition 3 *An implementation I is said to be wait-free if any invoke event Inv in every possible history H satisfies one of the followings.*

- *There exists a corresponding response event.*
- *For the process p_i in which Inv occurs, $Stop(p_i)$ occurs after Inv.* ■

The efficiency of an implementation I is measured by the worst-case response times of operation executions. For an operation execution (Inv, Res), we define the response time as $time(Res) - time(Inv)$. Let $OPE(H)$ denote a set of operation executions that appears in a history H. For an operation execution $ope = (Inv, Res)$, let $ope.op$ denote an operation invoked in Inv. For an implementation I of an object O supporting an operation op, we define the worst-case response time of op, denoted by $res\_time(op)$, as $\max\{res\_time(ope) | ope \in OPE(H), ope.op = op, H \text{ is a history of } I\}$.

3 Read/write registers

In this section, we present four implementations of a read/write register. We show the type of a read/write register on a domain D in Figure 2. The efficiency of a read/write register is measured by $res\_time(read)$ and $res\_time(write)$ where $write$ is any write operation $write(v)$.

$$
\begin{aligned}
&OP = \{write(v) | v \in D\} \cup \{read\} \\
&RES = \{\text{ack}\} \cup D \\
&Q = D \\
&\forall v, v' \quad \delta(v, write(v')) = (v', \text{ack}) \\
&\forall v \quad \delta(v, read) = (v, v)
\end{aligned}
$$

Figure 2: Type of a read/write register on on a domain D.

In all implementations, each process keeps a local copy of a read/write register. When a write operation is invoked at a process, the process assigns a timestamp to the write operation and broadcasts an update message that contains the written value and the timestamp of the write operation. A process updates its local copy according to received update messages. The update depends on a timestamp assigned to the write operation execution. In a read operation execution, the value of its local copy at some time during the operation is returned. For a read operation execution R, let $Write(R)$ be the write operation execution whose written value R returns.

We describe each program code by event-driven form for input events. A series of each event and the succeeding internal changes of the state is atomic, that is, the process does not crash during the series. If multiple input events occur at the same time, they are handled in an order such that they appear in the described code except a stop event.

3.1 Implementations using reliable broadcast

3.1.1 Asynchronous clocks

The first implementation is on a reliable broadcast and asynchronous clock model, which we call $register_{RB-AC}$ (for "reliable broadcast, asynchronous clocks"). The program code for p_i is given in Fig.3.

In a write operation execution, the process broadcasts an update message on the invocation. When a process receives the update message, it updates its local copy according to the message. The write operation execution is completed by returning ack after d since the invocation. In a read operation execution, the process decides the returned value for the value of its local copy on the

data type
timestamp=(integer, process identifier)
variables
count, **type** integer, **init** 0
res_val, **type** value of the register
last_up_ts, **type** timestamp, **init** $(0,0)$
local_copy, **type** value of the register
transition functions of process p_i
Invoke$(p_i, write(v))$:
  $count := count + 1$;
  BroadCast$(p_i,$ update$(v,(count,i)))$;
  /* update message */
  TimerSet$(p_i, d,$ WRITE);
Invoke$(p_i, read)$:
  $res\_val := local\_copy$;
  TimerSet$(p_i, u,$ READ);
Receive$(p_i, p_j,$ update$(v,(recvd_ct, recvd_uid)))$:
  $count :=$ max$(count, recvd_ct)$;
  if $last\_up\_ts <^4 (recvd\_ct, recvd\_uid)$
    then $local\_copy := v$;
      $last\_up\_ts := (recvd\_ct, recvd\_uid)$;
Alarm$(p_i,$ WRITE) :
  Response$(p_i,$ ack);
Alarm$(p_i,$ READ) :
  Response(p_i, res_val);
Stop(p_i) :
  No events can happen after this event.

Figure 3: $register_{RB-AC}$ (for p_i).

invocation, and it returns the value after u since it is invoked.

In this implementation, a monotone increasing integer *count* is used as a timestamp. A process increase *count* by 1 when it invokes a write operation. If the timestamp contained in a received update message is greater than the process's *count* (breaking tie by process identifiers), the process sets its *count* to the timestamp. Since any message delay is not greater than d, a write operation execution W_2 succeeding another write operation execution W_1 is assigned a greater timestamp than W_1. A process ignores an update message containing smaller timestamp than the last handled timestamp. In this case, the process considers that such an update message was handled and the value was overwritten by some write operation. An update message broadcasted at time t of a write operation execution W is received in $[t+d-u, t+d]$. Therefore, at each process, the update message for W is handled in this interval, or it is ignored. For two read operation executions R and R' such that R' precedes R, it is guaranteed that R returns the value written by the write operation with timestamp greater than or equal to R'.

We show that any possible history H in $register_{RB-AC}$ is linearizable and wait-free. A pending invoke event of a write operation W is said to be *valid* if there exists a read operation execution R such that $Write(R) = W$. Let H' be a history in which response events corresponding to valid pending events are appended to H in arbitrary order. We construct a legal sequence τ as follows. First, we assume that a sequence τ begins with a virtual write operation W_0 that writes the initial value, and arrange all write operation executions in $complete(H')$ after W_0 in order of their timestamps. Next, we put read operation executions that returns a written value of a write operation execution W immediately before W in order of their global invocation time. We can show that op_1 precedes op_2 in τ if $op_1 \overset{complete(H')}{\longrightarrow} op_2$, for any operation executions op_1 and op_2. Therefore, the following theorem holds.

Theorem 1 *The implementation $register_{RB-AC}$ is a wait-free linearizable implementation of a read/write register which achieves $res\_time(write) = d$ and $res\_time\ (read) = u$ on a reliable broadcast and an asynchronous clock model.* ■

3.1.2 u-synchronous clock

We present an implementation $register_{RB-uC}$ on a reliable broadcast and u-synchronous clock model (Fig.4). The implementation $register_{RB-uC}$ is a parameterized implementation with a parameter α ($leq\alpha \leq 1$) where $res\_time(write) \geq u$, $res\_time(write) + res\_time(read) \geq d$ and $res\_time(read) \geq u$ hold.

The implementation $register_{RB-uC}$ uses a local clock value as a timestamp instead of *count*. This guarantees that a preceding write operation execution has a smaller timestamp. A returned value of a read operation execution is decided at

[4]The symbol $<$ denotes lexicographic order. A relation $(a_1, b_1) < (a_2, b_2)$ implies that $a_1 < a_2$, or $a_1 = a_2$ and $b_1 < b_2$.

constant
$|W| = u + \alpha \cdot \max\{d - 2u, 0\}$,
$|R| = u + (1 - \alpha)\max\{d - 2u, 0\}$
data type
timestamp=(time, process identifier);
variables
local_cl, **type** time, **init** 0;
res_val, **type** value of the register ;
last_up_ts, **type** timestamp, **init** $(0, 0)$;
local_copy, **type** value of the register,
init initial value of the register;
transition functions of process p_i
Invoke$(p_i, write(v))$:
ReadClock$(p_i, local\_cl)$;
BroadCast$(p_i, \text{update}(v, (local\_cl, i)))$;
/* update message */
TimerSet$(p_i, |W|, \text{WRITE})$;
Invoke$(p_i, read)$:
TimerSet$(p_i, \min\{|R|, d - u\}, \text{SET\_VAL})$;
TimerSet$(p_i, |R|, \text{READ})$;
Receive$(p_i, p_j, \text{update}(v, (recvd\_cl, recvd\_uid)))$:
if $last\_up\_ts < (recvd\_cl, recvd\_uid)$
then $local\_copy := v$;
$last\_up\_ts := (recvd\_cl, recvd\_uid)$;
Alarm(p_i, WRITE) :
$last\_up\_ts := write\_ts$;
Response(p_i, ack);
Alarm$(p_i, \text{SET\_VAL})$:
$res\_val := local\_copy$;
Alarm(p_i, READ) :
Response(p_i, res_val);
Stop(p_i) :
No events can happen after this event.

Figure 4: $register_{RB-uC}$ (for p_i).

$min\{d-u, u+(1-\alpha)max\{d-2u, 0\}\}$. This guarantees that, for any read operation execution R and any write operation execution W preceding R, $Write(R)$ does not have a smaller timestamp than W. Furthermore, for any read operation execution R' preceding R, it is guaranteed that $Write(R)$ does not have a smaller timestamp than $Write(R')$. From these facts, we can prove the following theorem.

Theorem 2 *The implementation* $register_{RB-uC}$ *is a wait-free linearizable implementation of a read/write register which achieves* $res\_time(write) = u + \alpha \cdot \max\{d - 2u, 0\}$ *and* $res\_time\ (read) = u + (1 - \alpha)\max\{d - 2u, 0\}$ *$(0 \leq \alpha \leq 1)$ on a reliable broadcast and u-synchronous clock model.* ■

3.2 Implementations without reliable broadcast

3.2.1 Asynchronous clocks

Here we present implementation $register_{UB-AC}$ on an unreliable broadcast and asynchronous clock model (Fig.5). This implementation is based on $register_{RB-AC}$. Note that a message broadcasted in an unreliable broadcast model is not guaranteed to be received by all correct processes if the sender crashes during its broadcast. A message which all correct processes do not receive is called to be *incompletely broadcasted.*

If the update message is incompletely broadcasted, some correct processes does not receive it. In this case, if some process receives such a message and returns a value written by it, another process that does not receive the message may violate linearizability. In $register_{UB-AC}$, a process executing a read operation relays an update message containing a return value of this read operation execution to other processes. A process broadcasts such an *additional* update message as soon as it decides a return value and waits for the response time of a write operation before it returns a response. If a read operation execution is completed, it means that the process does not get faulty during the operation execution and every correct process can receive the additional update message. This guarantees linearizability.

Theorem 3 *The implementation* $register_{UB-AC}$ *is a wait-free linearizable implementation of a read/write register which achieves* $res\_time\ (write) = d$ *and* $res\_time\ (read) = d$ *on an unreliable broadcast and asynchronous clock model.* ■

3.2.2 u-synchronous clocks

Here we show an implementation $register_{UB-uC}$ on a u-synchronous clock and unreliable broadcast model (Fig.6). This implementation is based on $register_{RB-uC}$ and a process executing read operation relays an update messages like $register_{UB-AC}$. To minimize response times, we set $\alpha = 0$ for $register_{RB-uC}$ and modify it in a similar fashion to $register_{UB-AC}$. Consequently, we can show the following theorem.

data type
timestamp=(integer, process identifier)
variables
count, **type** integer, **init** 0
res_val, **type** value of the register
last_up_ts, **type** timestamp, **init** $(0,0)$
local_copy, **type** value of the register
transition functions of process p_i
Invoke$(p_i, write(v))$:
  $count := count + 1$;
  for $j = 1$ **to** n
  /* broadcast an original update message */
    do *Send*$(p_i, p_j,$ update$(v, (count, i)))$;
  TimerSet$(p_i, d,$ WRITE);
Invoke$(p_i, read)$:
  $res\_val := local\_copy$;
  for $j = 1$ **to** n
  /* broadcast an additional update message */
    do *Send*$(p_i, p_j,$ update$(res_val, last_up_ts))$;
  TimerSet$(p_i, d,$ READ);
Receive$(p_i,$ update$(v, (recvd_ct, recvd_uid)))$:
  $count :=$ max$(count, recvd_ct)$;
  if $last\_up\_ts < (recvd\_ct, recvd\_uid)$
    then $local\_copy := v$;
      $last\_up\_ts := (recvd\_ct, recvd\_uid)$;
Alarm$(p_i,$ WRITE) :
  Return$(p_i,$ ack);
Alarm$(p_i,$ READ) :
  Return(p_i, res_val);
Stop(p_i) :
  No events can happen after this event.

Figure 5: $register_{UB-AC}$ (for p_i.)

data type
timestamp=(time, process identifier);
variables
$local_cl$, **type** time, **init** 0;
res_val, **type** value of the register ;
last_up_ts, **type** timestamp, **init** $(0,0)$;
local_copy, **type** value of the register,
  init initial value of the register;
transition functions of process p_i
Invoke$(p_i, write(v))$:
  ReadClock$(p_i, local\_cl)$;
  for $j = 1$ **to** n
  /* broadcast an original update message */
    do *Send*$(p_i, p_j,$ update$(v, (local_cl, i)))$;
  TimerSet$(p_i, u,$ WRITE);
Invoke$(p_i, read)$:
  TimerSet$(p_i, d - u,$ SET_VAL);
  TimerSet$(p_i, d,$ READ);
Receive$(p_i, p_j,$ update$(v, (recvd_cl, recvd_uid)))$:
  if $last\_up\_ts < (recvd\_cl, recvd\_uid)$
    then $local\_copy := v$;
      $last\_up\_ts := (recvd\_cl, recvd\_uid)$;
Alarm$(p_i,$ WRITE) :
  $last\_up\_ts := write\_ts$;
  Response$(p_i,$ ack);
Alarm$(p_i,$ SET_VAL) :
  $res\_val := local\_copy$;
  for $j = 1$ **to** n
  /* broadcast an additional update message */
    do *Send*$(p_i, p_j,$ update$(res_val, last_up_ts))$;
Alarm$(p_i,$ READ) :
  Response(p_i, res_val);
Stop(p_i) :
  No events can happen after this event.

Figure 6: $register_{UB-uC}$ (for p_i.)

Theorem 4 *The implementation* $register_{UB-uC}$ *is a wait-free linearizable implementation of a read/write register which achieves* $res\_time\ (write) = d$ *and* $res\_time\ (read) = u$ *on an unreliable and u-synchronous clock model.* ■

4 General objects using reliable broadcast

4.1 Asynchronous clocks

We previously presented a linearizable implementation of a general object on an asynchronous clock model, where we achieved $res\_time(op_a) = u$ and $res\_time(op_v) = 2d$[4]. However, the implementation is not wait-free in a sense that it does not tolerate a crash fault of a process. In this subsection, we slightly modify that implementation so as to guarantee wait-freedom in the case where a reliable broadcast is available. First, we briefly explain the implementation presented in [4], and then mention the modification to produce a wait-freedom linearizable implementation, which we call $general_{RB-AC}$.

In the implementation in [4], any val-type operation needs $2d$ since its invocation to obtain its response value, and any operation needs u since its invocation to return a response that guarantees linearizability. Each process applies invoked operations to the implemented object sequentially in some common order to all processes. The total order is decided as follows. When an operation

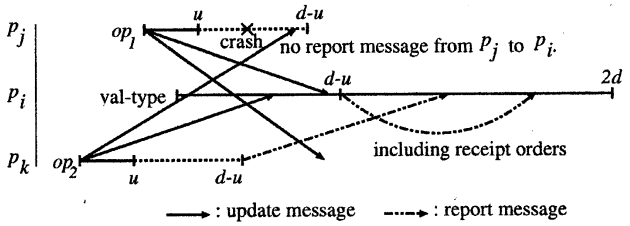

Figure 7: Case where a process crashes before broadcasting a report message.

op is invoked at a process p_i, p_i broadcasts an update message to inform of the invocation. After $d-u$ since this invocation, p_i regards the operation whose update messages p_i received before this time as operations prior to op in the total order, and broadcasts this order by a report message. We proved that a collection of such precedence relation forms surely a partial order and every process can obtain a common total order by extending it locally in a common rule. We also showed that a subset of the total order up to an operation op can be decided after $2d$ since its invocation. Therefore, the process can obtain its response value at that time. In the case of an ack-type operation, since its response value is unique, the process does not need $2d$ to obtain a response value but needs u for linearizability.

Now we modify this implementation for wait-freedom. Only the problem is the case where some process crashes soon after some ack-type operation op_1 completed in the process. If the process crashes before broadcasting the corresponding report message, any other process is not informed of the precedence relation about this operation. If some operation execution op_2 precedes another operation execution op_1 as in Fig.7, every process including p_i receives an update message of op_2 prior to an update message of op_1. In $general_{RB-AC}$, processes broadcast such receipt orders in their report messages. When a process applies operations to its local copy, if some report message brings that an update message of op_2 is received before one of op_1 and no report message brings the reverse, the process applies op_2 prior to op_1. This modification can achieve wait-freedom without additional response time.

Theorem 5 *The implementation* $general_{RB-AC}$ *is a wait-free linearizable implementation of any deterministic object which achieves* $res\_time(op_a) = u$ *and* $res\_time(op_v) = 2d$ *on a reliable broadcast and asynchronous clock model.* ■

4.2 u-synchronous clocks

Next we propose an implementation $general_{RB-uC}$ of a general object on u-synchronous clock model.

In this implementation, the common order to all processes is decided by a timestamp assigned to each operation. When an operation op is invoked at p_i, p_i assigns the value of its local clock as a timestamp to op, and broadcasts an update message with the timestamp. When a process receives an update message, it stores the information in its $update\_buffer$. Since the difference between any pair of local clock values is at most u and message delays are at most d, the process does not receive an update message with smaller timestamp than an operation op after $d+u$ since the invocation of the operation op. Therefore, if op is val-type, p_i can decide the total order of operations with smaller timestamp at that time and obtain its response value. And then, it returns the response. For an ack-type operation, the process need not obtain its returned value but need u for linearizability. If a process crashes while an operation, the operation is left pending. In such a case, all correct processes receive the update message, or no processes receive it because of a reliable broadcast. Therefore, the implementation $general_{RB-uC}$ works correctly in the case where a process crashes.

Theorem 6 *The implementation* $general_{RB-uC}$ *is a wait-free linearizable implementation of any deterministic object which achieves* $res\_time(op_a) = u$ *and* $res\_time(op_v) = d+u$ *on a reliable broadcast and u-synchronous clocks model.* ■

5 Conclusions

In this paper, we have presented the first wait-free linearizable implementations on a synchronous message passing system. We have considered four types of models on exchange of message and local clocks, and have presented four implemen-

tations of read/write registers, which are reliable/unreliable broadcast and asynchronous/u-synchronous clock models, and two implementations of general objects using reliable broadcasts on asynchronous/u-synchronous clock models (See Tab2).

In general, an implementation on an asynchronous clock model needs longer worst-case response times than an implementation on a u-synchronous clock model if the other conditions are the same. In an asynchronous clock model, if processes in the system execute a synchronization procedure (e.g. procedure Synch[7] for a reliable broadcast model) to make the difference between any pair of local clock values at most u, we can apply an implementation for a u-synchronous clock model. Taking costs of the synchronization procedure into consideration, implementations for a u-asynchronous clock model is more effective in the case where operations are invoked many times in an asynchronous clock model.

Some open problems are left. Some lower bound results as to worst-case response times in linearizable implementations were presented[2, 3, 4] . There are gaps between their results and our results. The other open problem is about linearizable implementations of general objects on an unreliable broadcast model. We can easily construct a wait-free linearizable implementation which provides operations with response time proportional to the number of processes. However, we do not know whether there exists a wait-free linearizable implementation which provides operations with shorter response time.

References

[1] M. Herlihy and J. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transaction on Programming Languages and Systems*, 12(3):463–492, 1990.

[2] H. Attiya and J. L. Welch. Sequential consistency versus linearizability. *ACM Transactions on Computer Systems*, 12(2):91–122, May 1994.

[3] M. Mavronicolas and D. Roth. Efficient, strongly consistent implementations of shared memory. *Proceedings of the 6th International Workshop on Distributed Algorithms(LNCS647)*, pages 346–361, 1992.

[4] M. Inoue, T. Masuzawa, and N. Tokura. Efficient linearizable implementation of shared fifo queues and general objects on a distributed system. *IEICE Transactions on Fundamentals on Electronics, Communications and Computer Sciences*, E81-A(5):768–775, May 1998.

[5] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, 1991.

[6] J. James and A. K. Singh. Fault tolerance bounds for memory consistency. *Proceedings of the 11th International Workshop on Distributed Algorithms (LNCS1320)*, pages 200–214, 1997.

[7] M. Mavronicolas and D. Roth. Sequential consistency and lineariability:read/write objects. *Proceedings of the 29th Annual Allerton Conference on Communication, Control and Computing*, Oct. 1991.